The SSL Protocol

Version 3.0

Internet Draft

March 1996 (Expires 9/96)

Alan O. Freier, Netscape Communications
Philip Karlton, Netscape Communications
Paul C. Kocher, Independent Consultant

Table of Contents

Appendices

SSL Version 3.0 - March 1996

1. Status of this memo

This document is an Internet-Draft. Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or made obsolete by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as *work in progress.*

To learn the current status of any Internet-Draft, please check the *1id-abstracts.txt* listing contained in the Internet Drafts Shadow Directories on ds.internic.net (US East Coast), nic.nordu.net (Europe), ftp.isi.edu (US West Coast), or munnari.oz.au (Pacific Rim).

---

2. Abstract

This document specifies Version 3.0 of the Secure Sockets Layer (SSL V3.0) protocol, a security protocol that provides communications privacy over the Internet. The protocol allows client/server applications to communicate in a way that is designed to prevent eavesdropping, tampering, or message forgery.

---

3. Introduction

The primary goal of the SSL Protocol is to provide privacy and reliability between two communicating applications. The protocol is composed of two layers. At the lowest level, layered on top of some reliable transport protocol (e.g., TCP[TCP]), is the **SSL Record Protocol**. The SSL Record Protocol is used for encapsulation of various higher level protocols. One such encapsulated protocol, the **SSL Handshake Protocol**, allows the server and client to authenticate each other and to negotiate an encryption algorithm and cryptographic keys before the application protocol transmits or receives its first byte of data. One advantage of SSL is that it is application protocol independent. A higher level protocol can layer on top of the SSL Protocol transparently.

The SSL protocol provides *connection security* that has three basic properties:

- The connection is private. Encryption is used after an initial handshake to define a secret key. Symmetric cryptography is used for data encryption (e.g., DES[DES], RC4[RC4], etc.)
- The peer's identity can be authenticated using asymmetric, or public key, cryptography (e.g., RSA[RSA], DSS[DSS], etc.).
- The connection is reliable. Message transport includes a message integrity check using a keyed MAC. Secure hash functions (e.g., SHA, MD5, etc.) are used for MAC computations.

---

4. Goals

The goals of SSL Protocol v3.0, in order of their priority, are:

1. Cryptographic security
SSL should be used to establish a secure connection between two parties.

2. Interoperability
Independent programmers should be able to develop applications utilizing SSL 3.0 that will then be able to successfully exchange cryptographic parameters without knowledge of one another's code.

Note: It is not the case that all instances of SSL (even in the same application domain) will be able to successfully connect. For instance, if the server supports a particular hardware token, and the client does not have access to such a token, then the connection will not succeed.

3. Extensibility
SSL seeks to provide a framework into which new public key and bulk encryption methods can be incorporated as necessary. This will also accomplish two sub-goals: to prevent the need to create a new protocol (and risking the introduction of possible new weaknesses) and to avoid the need to implement an entire new security library.

4. Relative efficiency
Cryptographic operations tend to be highly CPU intensive, particularly public key operations. For this reason, the SSL protocol has incorporated an optional session caching scheme to reduce the number of connections that need to be established from scratch. Additionally, care has been taken to reduce network activity.

5. Goals of this document

The SSL Protocol Version 3.0 Specification is intended primarily for readers who will be implementing the protocol and those doing cryptographic analysis of it. The spec has been written with this in mind, and it is intended to reflect the needs of those two groups. For that reason, many of the algorithm-dependent data structures and rules are included in the body of the text (as opposed to in an Appendix), providing easier access to them.

This document is not intended to supply any details of service definition nor interface definition, although it does cover select areas of policy as they are required for the maintenance of solid security.

6. Presentation language

This document deals with the formatting of data in an external representation. The following very basic and somewhat casually defined presentation syntax will be used. The syntax draws from several sources in its structure. Although it resembles the programming language 'C' in its syntax and **XDR** [XDR] in both its syntax and intent, it would be risky to draw too many parallels. The purpose of this presentation language is to document SSL only, not to have general application beyond that particular goal.

6.1 Basic block size

The representation of all data items is explicitly specified. The basic data block size is one byte (i.e. 8 bits). Multiple byte data items are concatenations of bytes, from left to right, from top to bottom. From the bytestream a multi-byte item (a numeric in the example) is formed (using C notation) by:

```
value = (byte[0] << 8*(n-1)) | (byte[1] << 8*(n-2)) | ... | byte[n-1];
```

This byte ordering for multi-byte values is the commonplace network byte order or big endian format.

6.2 Miscellaneous

Comments begin with "/*" and end with "*/".

Optional components are denoted by enclosing them in italic "*[ ]*" brackets.

Single byte entities containing uninterpreted data are of type `opaque`.

6.3 Vectors

A vector (single dimensioned array) is a stream of homogeneous data elements. The size of the vector may be specified at documentation time or left unspecified until runtime. In either case the length declares the number of bytes, not the number of elements, in the vector.

The syntax for specifying a new type *T'* that is a fixed length vector of type *T* is

```
T T'[n];
```

Here *T'* occupies *n* bytes in the data stream, where *n* is a multiple of the size of *T*. The length of the vector is not included in the encoded stream.

In the following example, `Datum` is defined to be three consecutive bytes that the protocol does not interpret, while `Data` is three consecutive `Datum`, consuming a total of nine bytes.

```
opaque Datum[3]; /* three uninterpreted bytes of data */
Datum Data[9];   /* 3 consecutive 3 byte vectors */
```

Variable length vectors are defined by specifying a subrange of legal lengths, inclusively, using the notation `<floor..ceiling>`. When encoded, the *actual length* precedes the vector's contents in the byte stream. The length will be in the form of a number consuming as many bytes as required to hold the vector's specified maximum (*ceiling*) length. A

variable length vector with an actual length field of zero is referred to as an empty vector.

```
T T'<floor..ceiling>;
```

In the following example, `mandatory` is a vector that must contain between 300 and 400 bytes of type `opaque`. It can never be empty. The *actual length* field consumes two bytes, a `uint16`, sufficient to represent the value 400 (see Section 6.4). On the other hand, longer can represent up to 800 bytes of data, or 400 uint16 elements, and it may be empty. Its encoding will include a two byte *actual length* field prepended to the vector.

```
opaque mandatory<300..400>; /* length field is 2 bytes, cannot be empty */
uint16 longer<0..800>;      /* zero to 400 16-bit unsigned integers */
```

6.4 Numbers

The basic numeric data type is an unsigned byte (`uint8`). All larger numeric data types are formed from fixed length series of bytes concatenated as described in Section 6.1 and are also unsigned. The following numeric types are predefined.

```
uint8 uint16[2];
uint8 uint24[3];
uint8 uint32[4];
uint8 uint64[8];
```

6.5 Enumerateds

An additional sparse data type is available called `enum`. A field of type `enum` can only assume the values declared in the definition. Each definition is a different type. Only enumerateds of the same type may be assigned or compared. Every element of an enumerated must be assigned a value, as demonstrated in the following example. Since the elements of the enumerated are not *ordered*, they can be assigned any unique value, in any order.

```
enum { e1(v1), e2 (v1), ... , en (vN), [(n)] } Te;
```

Enumerateds occupy as much space in the byte stream as would its maximal defined ordinal value. The following definition would cause one byte to be used to carry fields of type `Color`.

```
enum { red(3), blue(5), white(7) } Color;
```

One may optionally specify a value without its associated tag to force the width definition without defining a superfluous element. In the following example, `Taste` will consume two bytes in the data stream but can only assume the values 1, 2 or 4.

```
enum { sweet(1), sour(2), bitter(4), (32000) } Taste;
```

The names of the elements of an enumeration are scoped within the defined type. In the first example, a fully qualified reference to the second element of the enumeration would be `Color.blue`. Such qualification is not required if the target of the assignment is well specified.

```
Color color = Color.blue;       /* overspecified, but legal */
Color color = blue;     /* correct, type is implicit */
```

For enumerateds that are never converted to external representation, the numerical information may be omitted.

```
enum { low, medium, high } Amount;
```

6.6 Constructed types

Structure types may be constructed from primitive types for convenience. Each specification declares a new, unique type. The syntax for definition is much like that of C.

```
struct {
        T1 f1;
        T2 f2;
        ...
        Tn  fn;
} [T];
```

The fields within a structure may be qualified using the type's name using a syntax much like that available for enumerateds. For example, $T.f_2$ refers to the second field of the previous declaration. Structure definitions may be embedded.

6.6.1 Variants

Defined structures may have *variants* based on some knowledge that is available within the environment. The selector must be an enumerated type that defines the possible variants the structure defines. There must be a case arm for every element of the enumeration declared in the select. The body of the variant structure may be given a label for reference. The mechanism by which the variant is selected at runtime is not prescribed by the presentation language.

```
struct {
        T1 f1;
        T2 f2;
            ....
        Tn fn;
        select (E) {
            case e1: Te1;
            case e2: Te2;
                ....
            case en: Ten;
        } [fv];
} [Tv];
```

For example

```
enum { apple, orange } VariantTag;
struct {
        uint16 number;
        opaque string<0..10<;      /* variable length */
} V1;
struct {
        uint32 number;
        opaque string[10];       /* fixed length */
} V2;
struct {
        select (VariantTag) {    /* value of variant selector is implicit */
            case apple: V1;      /* definition of VariantBody, tag = apple */
            case orange: V2;     /* definition of VariantBody, tag = orange */
        } variant_body; /* optional label on the variant portion */
} VariantRecord;
```

Variant structures may be qualified (narrowed) by specifying a value for the selector prior to the type. For example, a

```
orange VariantRecord
```

is a *narrowed* type of a `VariantRecord` containing a `variant_body` of type V2.

6.7 Cryptographic attributes

The four cryptographic operations *digital signing, stream cipher encryption, block cipher encryption,* and *public key encryption* are designated `digitally-signed`, `stream-ciphered`, `block-ciphered`, and `public-key-encrypted`, respectively. A field's cryptographic processing is specified by prepending an appropriate key word designation before the field's type specification. Cryptographic keys are implied by the *current session state* (see Section 7.1).

In digital signing, one-way hash functions are used as input for a signing algorithm. In RSA signing, a 36-byte structure of two hashes (one SHA and one MD5) is *signed* (encrypted with the private key). In DSS, the 20 bytes of the SHA hash are run directly through the Digital Signing Algorithm with no additional hashing.

In stream cipher encryption, the plaintext is exclusive-ORed with an identical amount of output generated from a cryptographically-secure keyed pseudorandom number generator.

In block cipher encryption, every block of plaintext encrypts to a block of ciphertext. Because it is unlikely that the plaintext (whatever data is to be sent) will break neatly into the necessary block size (usually 64 bits), it is necessary to pad out the end of short blocks with some regular pattern, usually all zeroes.

In public key encryption, one-way functions with secret "trapdoors" are used to encrypt the outgoing data. Data encrypted with the public key of a given key pair can only be decrypted with the private key, and vice-versa.

In the following example:

```
stream-ciphered struct {
        uint8 field1;
        uint8 field2;
        digitally-signed opaque hash[20];
} UserType;
```

The contents of `hash` are used as input for a signing algorithm, then the entire structure is encrypted with a stream cipher.

6.8 Constants

Typed constants can be defined for purposes of specification by declaring a symbol of the desired type and assigning values to it. Under-specified types (`opaque`, variable length vectors, and structures that contain opaque) cannot be assigned values. No fields of a multi-element structure or vector may be elided.

For example,

```
struct {
        uint8 f1;
        uint8 f2;
} Example1;
Example1 ex1 = {1, 4};  /* assigns f1 = 1, f2 = 4 */
```

---

7. SSL protocol

SSL is a layered protocol. At each layer, messages may include fields for length, description, and content. SSL takes messages to be transmitted, fragments the data into manageable blocks, optionally compresses the data, applies a MAC, encrypts, and transmits the result. Received data is decrypted, verified, decompressed, and reassembled, then delivered to higher level clients.

7.1 Session and connection states

An SSL session is *stateful*. It is the responsibility of the SSL Handshake protocol to coordinate the states of the client and server, thereby allowing the protocol state machines of each to operate consistently, despite the fact that the state is not exactly parallel. Logically the state is represented twice, once as the current *operating* state, and (during the handshake protocol) again as the *pending* state. Additionally, separate read and write states are maintained. When the client or server receives a **change cipher spec** message, it copies the pending read state into the current read state. When the client or server sends a **change cipher spec** message, it copies the pending write state into the current write state. When the handshake negotiation is complete, the client and server exchange **change cipher spec** messages (see Section 7.3), and then communicate using the newly agreed-upon cipher spec.

An SSL session may include multiple secure connections; in addition, parties may have multiple simultaneous sessions.

The session state includes the following elements:

session identifier
: An arbitrary byte sequence chosen by the server to identify an active or resumable session state

peer certificate
: X509.v3[X509] certificate of the peer. This element of the state may be null.

compression method
: The algorithm used to compress data prior to encryption.

cipher spec
: Specifies the bulk data encryption algorithm (such as null, DES, etc.) and a MAC algorithm (such as MD5 or SHA). It also defines cryptographic attributes such as the `hash_size`. (See Appendix A.7 for formal definition.)

master secret
: 48-byte secret shared between the client and server.

is resumable
: A flag indicating whether the session can be used to initiate new connections.

The connection state includes the following elements:

server and client random
Byte sequences that are chosen by the server and client for each connection.
server write MAC secret
The secret used in MAC operations on data written by the server.
client write MAC secret
The secret used in MAC operations on data written by the client.
server write key
The bulk cipher key for data encrypted by the server and decrypted by the client.
client write key
The bulk cipher key for data encrypted by the client and decrypted by the server.
initialization vectors
When a block cipher in CBC mode is used, an initialization vector (IV) is maintained for each key. This field is first initialized by the SSL handshake protocol. Thereafter the final ciphertext block from each record is preserved for use with the following record.
sequence numbers
Each party maintains separate sequence numbers for transmitted and received messages for each connection. When a party sends or receives a **change cipher spec** message, the appropriate sequence number is set to zero. Sequence numbers are of type uint64 and may not exceed 2^{64}-1.

7.2 Record layer

The SSL Record Layer receives uninterpreted data from higher layers in non-empty blocks of arbitrary size.

7.2.1Fragmentation

The record layer fragments information blocks into SSLPlaintext records of 2^{14} bytes or less. Client message boundaries are not preserved in the record layer (i.e., multiple client messages of the same ContentType may be coalesced into a single SSLPlaintext record).

```
struct {
        uint8 major, minor;
} ProtocolVersion;
enum {
        change_cipher_spec(20), alert(21), handshake(22),
        application_data(23), (255)
} ContentType;
struct {
        ContentType type;
        ProtocolVersion version;
        uint16 length;
        opaque fragment[SSLPlaintext.length];
} SSLPlaintext;
```

`type`
The higher level protocol used to process the enclosed fragment.
`version`
The version of protocol being employed. This document describes SSL Version 3.0 (See Appendix A.1.1).
`length`
The length (in bytes) of the following SSLPlaintext.fragment. The length should not exceed 2^{14}.
`fragment`
The application data. This data is transparent and treated as an independent block to be dealt with by the higher level protocol specified by the type field.

Note: Data of different SSL Record layer content types may be interleaved. Application data is generally of lower precedence for transmission than other content types.

7.2.2 Record compression and decompression

All records are compressed using the compression algorithm defined in the current session state. There is always an active compression algorithm, however initially it is defined as `CompressionMethod.null`. The compression algorithm translates an SSLPlaintext structure into an SSLCompressed structure. Compression functions erase their state information whenever the CipherSpec is replaced.

Note: The CipherSpec is part of the session state described in Section 7.1. References to fields of the CipherSpec are made throughout this document using presentation syntax. A more complete description of the CipherSpec is shown in Appendix

A.7.

Compression must be lossless and may not increase the content length by more than 1024 bytes. If the decompression function encounters an `SSLCompressed.fragment` that would decompress to a length in excess of 2^{14} bytes, it should issue a fatal `decompression_failure` **alert** (Section 7.4.2).

```
struct {
        ContentType type;               /* same as SSLPlaintext.type */
        ProtocolVersion version;        /* same as SSLPlaintext.version */
        uint16 length;
        opaque fragment[SSLCompressed.length];
} SSLCompressed;
```

`length`
The length (in bytes) of the following `SSLCompressed.fragment`. The length should not exceed $2^{14} + 1024$.
`fragment`
The compressed form of `SSLPlaintext.fragment`.

Note: A `CompressionMethod.null` operation is an identity operation; no fields are altered. (See Appendix A.4.1)

Implementation note: Decompression functions are responsible for ensuring that messages cannot cause internal buffer overflows.

7.2.3 Record payload protection and the CipherSpec

All records are protected using the encryption and MAC algorithms defined in the current CipherSpec. There is always an active `CipherSpec`, however initially it is `SSL_NULL_WITH_NULL_NULL`, which does not provide any security.

Once the handshake is complete, the two parties have shared secrets which are used to encrypt records and compute keyed message authentication codes (MACs) on their contents. The techniques used to perform the encryption and MAC operations are defined by the `CipherSpec` and constrained by `CipherSpec.cipher_type`. The encryption and MAC functions translate an SSLCompressed structure into an `SSLCiphertext`. The decryption functions reverse the process. Transmissions also include a sequence number so that missing, altered, or extra messages are detectable.

```
struct {
        ContentType type;
        ProtocolVersion version;
        uint16 length;
        select (CipherSpec.cipher_type) {
            case stream: GenericStreamCipher;
            case block: GenericBlockCipher;
        } fragment;
} SSLCiphertext;
```

`type`
The type field is identical to `SSLCompressed.type`.

`version`
The version field is identical to `SSLCompressed.version`.

`length`
The length (in bytes) of the following `SSLCiphertext.fragment`. The length may not exceed $2^{14} + 2048$.

`fragment`
The encrypted form of `SSLCompressed.fragment`, including the MAC.

7.2.3.1 Null or standard stream cipher

Stream ciphers (including `BulkCipherAlgorithm.null` - See Appendix A.7) convert `SSLCompressed.fragment` structures to and from stream `SSLCiphertext.fragment` structures.

```
stream-ciphered struct {
          opaque content[SSLCompressed.length];
          opaque MAC[CipherSpec.hash_size];
} GenericStreamCipher;
```

The MAC is generated as:

```
hash(MAC_write_secret + pad_2 +
        hash (MAC_write_secret + pad_1 + seq_num + length + content));
```

where "+" denotes concatenation.

pad_1
: The character 0x36 repeated 48 time for MD5 or 40 times for SHA.

pad_2
: The character 0x5c repeated the same number of times.

seq_num
: The sequence number for this message.

hash
: The hashing algorithm derived from the cipher suite.

Note that the MAC is computed before encryption. The stream cipher encrypts the entire block, including the MAC. For stream ciphers that do not use a synchronization vector (such as RC4), the stream cipher state from the end of one record is simply used on the subsequent packet. If the CipherSuite is SSL_NULL_WITH_NULL_NULL, encryption consists of the identity operation (i.e., the data is not encrypted and the MAC size is zero implying that no MAC is used). SSLCiphertext.length is SSLCompressed.length plus CipherSpec.hash_size.

7.2.3.2 CBC block cipher

For block ciphers (such as RC2 or DES), the encryption and MAC functions convert SSLCompressed.fragment structures to and from block SSLCiphertext.fragment structures.

```
block-ciphered struct {
        opaque content[SSLCompressed.length];
        opaque MAC[CipherSpec.hash_size];
        uint8 padding[GenericBlockCipher.padding_length];
        uint8 padding_length;
} GenericBlockCipher;
```

The MAC is generated as described in Section 7.2.3.1.

padding
: Padding that is added to force the length of the plaintext to be a multiple of the block cipher's block length.

padding_length
: The length of the padding must be less than the cipher's block length and may be zero. The padding length should be such that the total size of the GenericBlockCipher structure is a multiple of the cipher's block length.

The encrypted data length (SSLCiphertext.length) is one more than the sum of SSLCompressed.length, CipherSpec.hash_size, and padding_length.

Note: With CBC block chaining the initialization vector (IV) for the first record is provided by the handshake protocol. The IV for subsequent records is the last ciphertext block from the previous record.

7.3 Change cipher spec protocol

The **change cipher spec** protocol exists to signal transitions in ciphering strategies. The protocol consists of a single message, which is encrypted and compressed under the *current* (not the *pending*) CipherSpec. The message consists of a single byte of value 1.

```
struct {
        enum { change_cipher_spec(1), (255) } type;
} ChangeCipherSpec;
```

The **change cipher spec** message is sent by both the client and server to notify the receiving party that subsequent records will be protected under the just-negotiated CipherSpec and keys.

Reception of this message causes the receiver to copy the *read pending* state into the *read current* state. Separate read and write states are maintained by both the SSL client and server. When the client or server receives a **change cipher spec** message, it copies the pending read state into the current read state. When the client or server writes a **change cipher spec** message, it copies the pending write state into the current write state. The client sends a **change cipher spec** message following handshake **key exchange** and **certificate verify** messages (if any), and the server sends one after successfully processing the **key exchange** message it received from the client. An unexpected **change cipher spec** message should generate an unexpected_message **alert** (Section 7.4.2). When resuming a previous session, the **change cipher spec** message is sent after the hello messages.

7.4 Alert protocol

One of the content types supported by the SSL Record layer is the *alert* type. **Alert** messages convey the severity of the message and a description of the alert. **Alert** messages with a level of fatal result in the immediate termination of the connection. In this case, other connections corresponding to the session may continue, but the session identifier must be invalidated, preventing the failed session from being used to establish new connections. Like other messages, **Alert** messages are encrypted and compressed, as specified by the current connection state.

```
enum { warning(1), fatal(2), (255) } AlertLevel;

enum {
        close_notify(0),
        unexpected_message(10),
        bad_record_mac(20),
        decompression_failure(30),
        handshake_failure(40), no_certificate(41), bad_certificate(42),
            unsupported_certificate(43), certificate_revoked(44),
            certificate_expired(45), certificate_unknown(46),
            illegal_parameter (47)
        (255)
} AlertDescription;

struct {
        AlertLevel level;
        AlertDescription description;
} Alert;
```

7.4.1 Closure alerts

The client and the server must share knowledge that the connection is ending in order to avoid a truncation attack. Either party may initiate the exchange of closing messages.

close_notify
: This message notifies the recipient that the sender will not send any more messages on this connection. The session becomes unresumable if any connection is terminated without proper close_notify messages with level equal to warning.

7.4.2 Error alerts

Error handling in the SSL Handshake protocol is very simple. When an error is detected, the detecting party sends a message to the other party. Upon transmission or receipt of an **fatal alert** message, both parties immediately close the connection. Servers and clients are required to forget any session-identifiers, keys, and secrets associated with a failed connection. The following error alerts are defined:

unexpected_message
: An inappropriate message was received. This alert is always fatal and should never be observed in communication between proper implementations.

bad_record_mac
: This alert is returned if a record is received with an incorrect MAC. This message is always fatal.

decompression_failure
: The decompression function received improper input (e.g. data that would expand to excessive length). This message is always fatal.

handshake_failure
Reception of a handshake_failure **alert** message indicates that the sender was unable to negotiate an acceptable set of security parameters given the options available. This is a fatal error.

no_certificate
A no_certificate **alert** message may be sent in response to a **certification request** if no appropriate certificate is available.

bad_certificate
A certificate was corrupt, contained signatures that did not verify correctly, etc.

unsupported_certificate
A certificate was of an unsupported type.

certificate_revoked
A certificate was revoked by its signer.

certificate_expired
A certificate has expired or is not currently valid.

certificate_unknown
Some other (unspecified) issue arose in processing the certificate, rendering it unacceptable.

illegal_parameter
A field in the handshake was out of range or inconsistent with other fields. This is always fatal.

7.5 Handshake protocol overview

The cryptographic parameters of the session state are produced by the **SSL Handshake Protocol**, which operates on top of the **SSL Record Layer**. When a SSL client and server first start communicating, they agree on a protocol version, select cryptographic algorithms, optionally authenticate each other, and use public-key encryption techniques to generate shared secrets. These processes are performed in the handshake protocol, which can be summarized as follows:

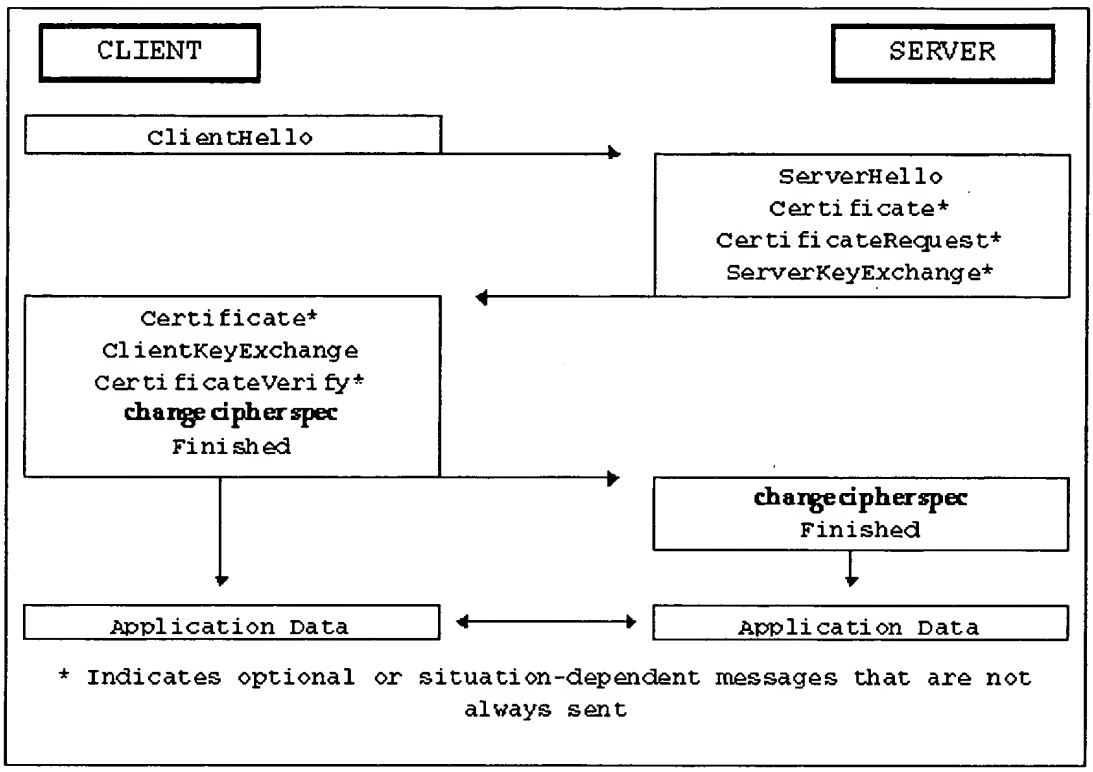
The client sends a **client hello** message to which the server must respond with a **server hello** message, or else a fatal error will occur and the connection will fail. The **client hello** and **server hello** are used to establish security enhancement capabilities between client and server. The **client hello** and **server hello** establish the following attributes: protocol version, session ID, cipher suite, and compression method. Additionally, two random values are generated and exchanged: ClientHello.random and ServerHello.random.

Following the hello messages, the server will send its certificate, if it is to be authenticated. Additionally, a **server key exchange** message may be sent, if it is required (e.g. if their server has no certificate, or if its certificate is for signing only). If the server is authenticated, it may request a certificate from the client, if that is appropriate to the cipher suite selected.

Now the server will send the **server hello done** message, indicating that the hello-message phase of the handshake is complete. The server will then wait for a client response.

If the server has sent a **certificate request** message, the client must send either the **certificate message** or a **no certificate** alert. The **client key exchange** message is now sent, and the content of that message will depend on the public key algorithm selected between the **client hello** and the **server hello**. If the client has sent a certificate with signing ability, a digitally-signed **certificate verify** message is sent to explicitly verify the certificate.

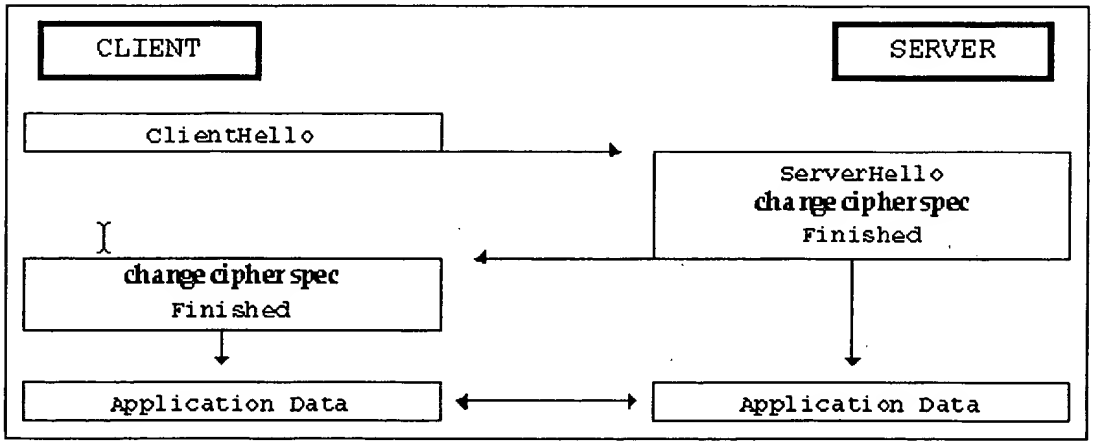
At this point, a **change cipher spec** message is sent by the client, and the client copies the *pending* Cipher Spec into the *current* Cipher Spec. The client then immediately sends the **finished** message under the new algorithms, keys, and secrets. In response, the server will send its own **change cipher spec** message, transfer the *pending* to the *current* Cipher Spec, and send its **Finished** message under the new Cipher Spec. At this point, the handshake is complete and the client and server may begin to exchange application layer data. (See flow chart below.)



Note: To help avoid pipeline stalls, ChangeCipherSpec is an independent SSL Protocol content type, and is not actually an SSL handshake message.

When the client and server decide to resume a previous session or duplicate an existing session (instead of negotiating new security parameters) the message flow is as follows:

The client sends a **client hello** using the Session ID of the session to be resumed. The Server then checks its session cache for a match. If a match is found, and the server is willing to re-establish the connection under the specified session state, it will send a **server hello** with the same Session ID value. At this point, both client and server must send **change cipher spec** messages and proceed directly to **finished** messages. Once the re-establishment is complete, the client and server may begin to exchange application layer data. (See flow chart below.) If a Session ID match is not found, the server generates a new session ID and the SSL client and server perform a full handshake.



The contents and significance of each message will be presented in detail in the following sections.

7.6 Handshake protocol

The SSL Handshake Protocol is one of the defined higher level clients of the SSL Record Protocol. This protocol is used to negotiate the secure attributes of a *session*. Handshake messages are supplied to the SSL Record Layer, where they are encapsulated within one or more SSLPlaintext structures, which are processed and transmitted as specified by the current active session state.

```
enum {
        hello_request(0), client_hello(1), server_hello(2),
        certificate(11), server_key_exchange (12), certificate_request(13),
        server_hello_done(14), certificate_verify(15), client_key_exchange(16),
        finished(20), (255)
} HandshakeType;

struct {
        HandshakeType msg_type;         /* type of handshake message */
        uint24 length;  /* # bytes in handshake message body */
        select (HandshakeType) {
            case hello_request: HelloRequest;
            case client_hello: ClientHello;
            case server_hello: ServerHello;
            case certificate: Certificate;
            case server_key_exchange: ServerKeyExchange;
            case certificate_request: CertificateRequest;
            case server_hello_done: ServerHelloDone;
            case certificate_verify: CertificateVerify;
            case client_key_exchange: ClientKeyExchange;
            case finished: Finished;
        } body;
} Handshake;
```

The handshake protocol messages are presented in the order they must be sent; sending handshake messages in an unexpected order results in a fatal error.

7.6.1 Hello messages

The hello phase messages are used to exchange security enhancement capabilities between the client and server. When a new session begins, the CipherSpec encryption, hash, and compression algorithms are initialized to null. The current CipherSpec is used for renegotiation messages.

7.6.1.1 Hello request

The **hello request** message may be sent by the server at any time, but will be ignored by the client if the handshake protocol is already underway. It is a simple notification that the client should begin the negotiation process anew by sending a **client hello** message when convenient.

Note: Since handshake messages are intended to have transmission precedence over application data, it is expected that the negotiation begin in no more than one or two times the transmission time of a maximum length application data message.

After sending a **hello request**, servers should not repeat the request until the subsequent handshake negotiation is complete. A client that receives a **hello request** while in a handshake negotiation state should simply ignore the message.

The structure of a **hello request** message is as follows:

```
struct { } HelloRequest;
```

7.6.1.2 Client hello

When a client first connects to a server it is required to send the **client hello** as its first message. The client can also send a **client hello** in response to a **hello request** or on its own initiative in order to renegotiate the security parameters in an existing connection.

The **client hello** message includes a random structure, which is used later in the protocol.

```
struct {
   uint32 gmt_unix_time;
   opaque random_bytes[28];
} Random;
```

gmt_unix_time
The current time and date in standard UNIX 32-bit format according to the sender's internal clock. Clocks are not required to be set correctly by the basic SSL Protocol; higher level or application protocols may define additional requirements.

random_bytes
28 bytes generated by a secure random number generator.

The **client hello** message includes a variable length session identifier. If not empty, the value identifies a session between the same client and server whose security parameters the client wishes to reuse. The session identifier may be from an earlier connection, this connection, or another currently active connection. The second option is useful if the client only wishes to update the random structures and derived values of a connection, while the third option makes it possible to establish several simultaneous independent secure connections without repeating the full handshake protocol. The actual contents of the SessionID are defined by the server.

```
opaque SessionID<0..32>;
```

Warning: Servers must not place confidential information in session identifiers or let the contents of fake session identifiers cause any breach of security.

The CipherSuite list, passed from the client to the server in the **client hello** message, contains the combinations of cryptographic algorithms supported by the client in order of the client's preference (first choice first). Each CipherSuite defines both a key exchange algorithm and a CipherSpec. The server will select a cipher suite or, if no acceptable choices are presented, return a **handshake failure** alert and close the connection.

```
uint8 CipherSuite[2];    /* Cryptographic suite selector */
```

The **client hello** includes a list of compression algorithms supported by the client, ordered according to the client's preference. If the server supports none of those specified by the client, the session must fail.

```
enum { null(0), (255) } CompressionMethod;
```

Issue: Which compression methods to support is under investigation.

The structure of the **client hello** is as follows.

```
struct {
        ProtocolVersion client_version;
        Random random;
        SessionID session_id;
        CipherSuite cipher_suites<2..2^16-1>;
        CompressionMethod compression_methods<1..2^8-1>;
} ClientHello;
```

client_version
The version of the SSL protocol by which the client wishes to communicate during this session. This should be the most recent (highest valued) version supported by the client. For this version of the specification, the version will be 3.0 (See Appendix E for details about backward compatibility).

random
A client-generated random structure.

session_id
The ID of a session the client wishes to use for this connection. This field should be *empty* if no session_id is available or the client wishes to generate new security parameters.

cipher_suites
This is a list of the cryptographic options supported by the client, sorted with the client's first preference first. If the session_id field is not *empty* (implying a session resumption request) this vector must include at least the cipher_suite from that session. Values are defined in Appendix A.6.

compression_methods
This is a list of the compression methods supported by the client, sorted by client preference. If the session_id field is not *empty* (implying a session resumption request) this vector must include at least the compression_method from that session. All implementations must support CompressionMethod.null.

After sending the **client hello** message, the client waits for a **server hello** message. Any other handshake message returned by the server except for a **hello request** is treated as a fatal error.

Implementation note: Application data may not be sent before a **finished** message has been sent. Transmitted application data is known to be insecure until a valid **finished** message has been received. This absolute restriction is relaxed if there is a current, non-null encryption on this connection.

7.6.1.3 Server hello

The server processes the **client hello** message and responds with either a handshake_failure **alert** or **server hello** message.

```
struct {
        ProtocolVersion server_version;
        Random random;
        SessionID session_id;
        CipherSuite cipher_suite;
        CompressionMethod compression_method;
} ServerHello;
```

server_version
This field will contain the lower of that suggested by the client in the **client hello** and the highest supported by the server. For this version of the specification, the version will be 3.0 (See Appendix E for details about backward compatibility).

random
This structure is generated by the server and *must* be different from (and independent of) ClientHello.random.

session_id
This is the identity of the session corresponding to this connection. If the ClientHello.session_id was non-empty, the server will look in its session cache for a match. If a match is found and the server is willing to establish the new connection using the specified session state, the server will respond with the same value as was supplied by the client. This indicates a *resumed* session and dictates that the parties *must* proceed directly to the **finished** messages. Otherwise this field will contain a different value identifying the new session. The server may return an *empty* session_id to indicate that the session will not be cached and therefore cannot be resumed.

cipher_suite
The single cipher suite selected by the server from the list in ClientHello.cipher_suites. For *resumed* sessions this field is the value from the state of the session being resumed.

compression_method
The single compression algorithm selected by the server from the list in ClientHello.compression_methods. For *resumed* sessions this field is the value from the resumed session state.

7.6.2 Server certificate

If the server is to be authenticated (which is generally the case), the server sends its certificate immediately following the **server hello** message. The certificate type must be appropriate for the selected cipher suite's key exchange algorithm, and is generally an

X.509.v3 certificate (or a modified X.509 certificate in the case of Fortezza [FOR]). The same message type will be used for the client's response to a **server certificate request** message.

```
opaque ASN.1Cert<1..2^24-1>;

struct {
        ASN.1Cert certificate_list<1..2^24-1>;
} Certificate;
```

certificate_list This is a sequence (chain) of X.509.v3 certificates, ordered with the sender's certificate first and the root certificate authority last.

Note: PKCS #7 [PKCS7] is not used as the format for the certificate vector because PKCS #6 [PKCS6] extended certificates are not used. Also PKCS #7 defines a SET rather than a SEQUENCE, making the task of parsing the list more difficult.

7.6.3 Server key exchange message

The **server key exchange** message is sent by the server if it has no certificate, has a certificate only used for signing (e.g., DSS [DSS] certificates, signing-only RSA [RSA] certificates), or fortezza/DMS key exchange is used. This message is not used if the server certificate contains Diffie-Hellman [DH1] parameters.

Note: According to current US export law, RSA moduli larger than 512 bits may not be used for key exchange in software exported from the US. With this message, larger RSA keys may be used as signature-only certificates to sign temporary shorter RSA keys for key exchange.

```
enum { rsa, diffie_hellman, fortezza_dms } KeyExchangeAlgorithm;

struct {
        opaque rsa_modulus<1..2^16-1>;
        opaque rsa_exponent<1..2^16-1>;
} ServerRSAParams;
```

rsa_modulus The modulus of the server's temporary RSA key.

rsa_exponent The public exponent of the server's temporary RSA key.

```
struct {
        opaque dh_p<1..2^16-1>;
        opaque dh_g<1..2^16-1>;
        opaque dh_Ys<1..2^16-1>;
} ServerDHParams;       /* Ephemeral DH parameters */
```

dh_p
The prime modulus used for the Diffie-Hellman operation.

dh_g
The generator used for the Diffie-Hellman operation.

dh_Y_s
The server's Diffie-Hellman public value (g^X mod p).

```
struct {
           opaque r_s [128];
} ServerFortezzaParams;
```

r_s
Server random number for Fortezza KEA (Key Exchange Algorithm).

```
struct {
        select (KeyExchangeAlgorithm) {
            case diffie_hellman:
                  ServerDHParams params;
                  Signature signed_params;
            case rsa:
                  ServerRSAParams params;
```

```
                Signature signed_params;
            case fortezza_dms:
                ServerFortezzaParams params;
        };
} ServerKeyExchange;
```

params
The server's key exchange parameters.

signed_params
A hash of the corresponding params value, with the signature appropriate to that hash applied.

md5_hash
MD5(ClientHello.random + ServerHello.random + ServerParams);

sha_hash
SHA(ClientHello.random + ServerHello.random + ServerParams);

```
enum { anonymous, rsa, dsa } SignatureAlgorithm;

digitally-signed struct {
        select(SignatureAlgorithm) {
            case anonymous: struct { };
            case rsa:
                opaque md5_hash[16];
                opaque sha_hash[20];
            case dsa:
                opaque sha_hash[20];
        };
} Signature;
```

7.6.4 Certificate request

A non-anonymous server can optionally request a certificate from the client, if appropriate for the selected cipher suite.

```
opaque CertificateAuthority<0..2^24-1>;

enum {
        rsa_sign(1), dss_sign(2), rsa_fixed_dh(3), dss_fixed_dh(4),
        rsa_ephemeral_dh(5), dss_ephemeral_dh(6), fortezza_dms(20), (255)
} ClientCertificateType;

opaque DistinguishedName<1..2^16-1>;

struct {
        ClientCertificateType certificate_types<1..2^8-1>;
        DistinguishedName certificate_authorities<3..2^16-1>;
} CertificateRequest;
```

certificate_types
This field is a list of the types of certificates requested, sorted in order of the server's preference.

certificate_authorities
A list of the distinguished names of acceptable certificate authorities.

Note: DistinguishedName is derived from [X509].

Note: It is a fatal handshake_failure **alert** for an anonymous server to request client identification.

7.6.5 Server hello done

The **server hello done** message is sent by the server to indicate the end of the **server hello** and associated messages. After sending this message the server will wait for a client response.

```
struct { } ServerHelloDone;
```

Upon receipt of the **server hello done** message the client should verify that the server provided a valid certificate if required and check that the **server hello** parameters are acceptable.

7.6.6 Client certificate

This is the first message the client can send after receiving a **server hello done** message. This message is only sent if the server requests a certificate. If no suitable certificate is available, the client should send a **no certificate** alert instead. This error is only a warning, however the server may respond with a **fatal handshake failure** alert if client authentication is required.

Client certificates are sent using the Certificate defined in Section 7.6.2.

Note: Client Diffie-Hellman certificates must match the server specified Diffie-Hellman parameters.

7.6.7 Client key exchange message

The choice of messages depends on which public key algorithm(s) has (have) been selected. See Section 7.6.3 for the KeyExchangeAlgorithm.

```
struct {
        select (KeyExchangeAlgorithm) {
            case rsa: EncryptedPreMasterSecret;
            case diffie_hellman: ClientDiffieHellmanPublic;
            case fortezza_dms: FortezzaKeys;
        } exchange_keys;
} ClientKeyExchange;
```

The information to select the appropriate record structure is in the *pending* session state (see Section 7.1).

7.6.7.1 RSA encrypted premaster secret message

If RSA is being used for key agreement and authentication, the client generates a 48-byte pre-master secret, encrypts it under the public key from the server's certificate or temporary RSA key from a **server key exchange** message, and sends the result in an **encrypted premaster secret** message.

```
struct {
        ProtocolVersion client_version;
        opaque random[46];
} PreMasterSecret;
```

client_version
: The latest (newest) version supported by the client. This is used to detect version roll-back attacks.

random
: 46 securely-generated random bytes.

```
struct {
        public-key-encrypted PreMasterSecret pre_master_secret;
} EncryptedPreMasterSecret;
```

pre_master_secret
: This random value is generated by the client and is used to generate the *master secret*, as specified in Section 8.1.

7.6.7.2 Fortezza key exchange message

Under Fortezza DMS, the client derives a Token Encryption Key (TEK) using Fortezza's Key

Exchange Algorithm (KEA). The client's KEA calculation uses the public key in the server's certificate along with private parameters in the client's token. The client sends public parameters needed for the server to generate the TEK, using its own private parameters. The client generates session keys, wraps them using the TEK, and sends the results to the server. The client generates IV's for the session keys and TEK and sends them also. The client generates a random 48-byte premaster secret, encrypts it using the TEK, and sends the result:

```
struct {
        opaque y_c<0..128>;
        opaque r_c[128];
        opaque y_signature[20];
        opaque wrapped_client_write_key[12];
        opaque wrapped_server_write_key[12];
        opaque client_write_iv[24];
        opaque server_write_iv[24];
        opaque master_secret_iv[24];
        block-ciphered opaque encrypted_pre_master_secret[48];
} FortezzaKeys;
```

y_signature
: y_singnature is the signature of the KEA public key, signed with the client's DSS private key.

y_c
: The client's Y_c value (public key) for the KEA calculation. If the client has sent a certificate, and its KEA public key is suitable, this value must be *empty* since the certificate already contains this value. If the client sent a certificate without a suitable public key, y_c is used and y_singnature is the KEA public key signed with the client's DSS private key. For this value to be used, it must be between 64 and 128 bytes.

r_c
: The client's R_c value for the KEA calculation.

wrapped_client_write_key
: This is the client's write key, wrapped by the TEK.

wrapped_server_write_key
: This is the server's write key, wrapped by the TEK.

client_write_iv
: This is the IV for the client write key.

server_write_iv
: This is the IV for the server write key.

master_secret_iv
: This is the IV for the TEK used to encrypt the pre-master secret.

pre_master_secret
: This is a random value, generated by the client and used to generate the *master secret*, as specified in Section 8.1. In the above structure, it is encrypted using the TEK.

7.6.7.3 Client Diffie-Hellman public value

This structure conveys the client's Diffie-Hellman public value (Y_c) if it was not already included in the client's certificate. The encoding used for Y_c is determined by the enumerated PublicValueEncoding.

```
enum { implicit, explicit } PublicValueEncoding;
```

implicit
: If the client certificate already contains the public value, then it is implicit and Y_c does not need to be sent again.

explicit
: Y_c needs to be sent.

```
struct {
        select (PublicValueEncoding) {
            case implicit: struct { };
            case explicit: opaque dh_Yc<1..2^16-1>;
        } dh_public;
} ClientDiffieHellmanPublic;
```

dh_Yc
The client's Diffie-Hellman public value (Y_c).

7.6.8 Certificate verify

This message is used to provide explicit verification of a client certificate. This message is only sent following any client certificate that has signing capability (i.e. all certificates except those containing fixed Diffie-Hellman parameters).

```
struct {
     Signature signature;
} CertificateVerify;
```

```
CertificateVerify.signature.md5_hash
        MD5(master_secret + pad2 + MD5(handshake_messages +
                master_secret + pad1));

Certificate.signature.sha_hash
        SHA(master_secret + pad2 + SHA(handshake_messages +
                master_secret + pad1));
```

Here handshake_messages refers to all handshake messages starting at **client hello** up to but not including this message.

7.6.9 Finished

A **finished** message is always sent immediately after a **change cipher specs** message to verify that the key exchange and authentication processes were successful. The **finished** message is the first protected with the just-negotiated algorithms, keys, and secrets. No acknowledgment of the **finished** message is required; parties may begin sending confidential data immediately after sending the **finished** message. Recipients of **finished** messages must verify that the contents are correct.

```
enum { client(0x434C4E54), server(0x53525652) } Sender;

struct {
        opaque md5_hash[16];
        opaque sha_hash[20];
} Finished;
```

```
md5_hash
        MD5(master_secret + pad2 + MD5(handshake_messages +
                Sender + master_secret + pad1));

sha_hash
        SHA(master_secret + pad2 + SHA(handshake_messages +
                Sender + master_secret + pad1));
```

The hash contained in **finished** messages sent by the server incorporate Sender.server; those sent by the client incorporate Sender.client. The value handshake_messages includes all handshake messages starting at **client hello** up to, but not including, the **finished** messages. This may be different from handshake_messages in Section 7.6.8 because it would include the **certificate verify** message (if sent).

Note: Change cipher spec messages are not handshake messages and are not included in the hash computations.

7.7 Application data protocol

Application data messages are carried by the Record Layer and are fragmented, compressed and encrypted based on the current connection state. The messages are treated as *transparent data* to the record layer.

---

8. Cryptographic computations

The key exchange, authentication, encryption, and MAC algorithms are determined by the cipher_suite selected by the server and revealed in the **server hello** message.

8.1 Asymmetric cryptographic computations

The asymmetric algorithms are used in the handshake protocol to authenticate parties and to generate shared keys and secrets.

For Diffie-Hellman, RSA, and Fortezza, the same algorithm is used to convert the pre_master_secret into the master_secret. The pre_master_secret should be deleted from memory once the master_secret has been computed.

```
master_secret =
       MD5(pre_master_secret + SHA('A' + pre_master_secret +
           ClientHello.random + ServerHello.random)) +
       MD5(pre_master_secret + SHA('BB' + pre_master_secret +
           ClientHello.random + ServerHello.random)) +
       MD5(pre_master_secret + SHA('CCC' + pre_master_secret +
           ClientHello.random + ServerHello.random));
```

8.1.1 RSA

When RSA is used for server authentication and key exchange, a 48-byte pre_master_secret is generated by the client, encrypted under the server's public key, and sent to the server. The server uses its private key to decrypt the pre_master_secret. Both parties then convert the pre_master_secret into the master_secret, as specified above.

RSA digital signatures are performed using PKCS #1 [PKCS1] block type 1. RSA public key encryption is performed using PKCS #1 block type 2.

8.1.2 Diffie-Hellman

A conventional Diffie-Hellman computation is performed. The negotiated key (Z) is used as the pre_master_secret, and is converted into the master_secret, as specified above.

Note: Diffie-Hellman parameters are specified by the server, and may be either ephemeral or contained within the server's certificate.

8.1.3 Fortezza

A random 48-byte pre_master_secret is sent encrypted under the TEK and its IV. The server decrypts the pre_master_secret and converts it into a master_secret, as specified above. Bulk cipher keys and IVs for encryption are generated by the client's token and exchanged in the **key exchange** message; the master_secret is only used for MAC computations.

8.2 Symmetric cryptographic calculations and the CipherSpec

The technique used to encrypt and verify the integrity of SSL records is specified by the currently active CipherSpec. A typical example would be to encrypt data using DES and generate authentication codes using MD5. The encryption and MAC algorithms are set to SSL_NULL_WITH_NULL_NULL at the beginning of the **SSL Handshake Protocol**, indicating that no message authentication or encryption is performed. The handshake protocol is used to negotiate a more secure CipherSpec and to generate cryptographic keys.

8.2.1 The master secret

Before secure encryption or integrity verification can be performed on records, the client and server need to generate shared secret information known only to themselves. This value is a 48-byte quantity called the *master secret*. The master secret is used to generate keys and secrets for encryption and MAC computations. Some algorithms, such as Fortezza, may have their own procedure for generating encryption keys (the master secret is used only for MAC computations in Fortezza).

8.2.2 Converting the master secret into keys and MAC secrets

The master secret is hashed into a sequence of secure bytes, which are assigned to the MAC secrets, keys, and non-export IVs required by the current CipherSpec (see Appendix A.7).

CipherSpecs require a *client write MAC secret, a server write MAC secret, a client write key, a server write key, a client write IV,* and a *server write IV,* which are generated from the master secret in that order. Unused values, such as Fortezza keys communicated in the KeyExchange message, are *empty*. The following inputs are available to the key definition process:

```
opaque MasterSecret[48]
ClientHello.random
ServerHello.random
```

When generating keys and MAC secrets, the master secret is used as an entropy source, and the random values provide unencrypted salt material and IVs for exportable ciphers.

To generate the key material, compute

```
key_block =
        MD5(master_secret + SHA('A' + master_secret + ServerHello.random +
            ClientHello.random)) +
        MD5(master_secret + SHA('BB' + master_secret + ServerHello.random +
            ClientHello.random)) +
        MD5(master_secret + SHA('CCC' + master_secret + ServerHello.random +
            ClientHello.random)) + [...];
```

until enough output has been generated. Then the key_block is partitioned as follows.

```
client_write_MAC_secret[CipherSpec.hash_size]
server_write_MAC_secret[CipherSpec.hash_size]
client_write_key[CipherSpec.key_material]
server_write_key[CipherSPec.key_material]
client_write_IV[CipherSpec.IV_size]                    /* non-export ciphers */
server_write_IV[CipherSpec.IV_size]                    /* non-export ciphers */
```

Any extra key_block material is discarded.

Exportable encryption algorithms (for which CipherSpec.is_exportable is true) require additional processing as follows to derive their final write keys:

```
final_client_write_key = MD5(client_write_key +
        ClientHello.random + ServerHello.random);
final_server_write_key = MD5(server_write_key +
        ServerHello.random + ClientHello.random);
```

Exportable encryption algorithms derive their IVs from the random messages:

```
client_write_IV = MD5(ClientHello.random + ServerHello.random);
server_write_IV = MD5(ServerHello.random + ClientHello.random);
```

MD5 outputs are trimmed to the appropriate size by discarding the least-significant bytes.

8.2.2.1 Export key generation example

SSL_RSA_EXPORT_WITH_RC2_CBC_40_MD5 requires five random bytes for each of the two encryption keys and 16 bytes for each of the MAC keys, for a total of 42 bytes of key material. MD5 produces 16 bytes of output per call, so three calls to MD5 are required. The MD5 outputs are concatenated into a 48-byte key_block with the first MD5 call providing bytes zero through 15,

the second providing bytes 16 through 31, etc. The key_block is partitioned, and the write keys are salted because this is an exportable encryption algorithm.

```
client_write_MAC_secret = key_block[0..15]
server_write_MAC_secret = key_block[16..31]
client_write_key        = key_block[32..36]
server_write_key        = key_block[37..41]

final_client_write_key = MD5 (client_write_key +
        ClientHello.random + ServerHello.random) [0..15];
final_server_write_key = MD5 (server_write_key +
        ServerHello.random + ClientHello.random) [0..15];

client_write_IV = MD5(ClientHello.random + ServerHello.random) [0..7];
server_write_IV = MD5(ServerHello.random + ClientHello.random) [0..7];
```